RIACS

Research Institute for Advanced Computer Science
NASA Ames Research Center

Improved Load Distribution in Parallel Sparse Cholesky Factorization

Edward Rothberg
Robert Schreiber

RIACS Technical Report 94.13 **August, 1994**

To appear: *Proceedings of Supercomputing 94*

Improved Load Distribution in Parallel Sparse Cholesky Factorization

Edward Rothberg
Robert Schreiber

The Research Institute of Advanced Computer Science is operated by Universities Space Research Association, The American City Building, Suite 212, Columbia, MD 21044, (410) 730-2656

Work reported herein was supported by NASA via Contract NAS 2-13721 between NASA and the Universities Space Research Association (USRA). Work was performed at the Research Institute for Advanced Computer Science (RIACS), NASA Ames Research Center, Moffett Field, CA 94035-1000.

Improved Load Distribution in Parallel Sparse Cholesky Factorization

Edward Rothberg

Intel Supercomputer Systems Division
14924 N.W. Greenbrier Parkway
Beaverton, OR 97006

Robert Schreiber

Research Institute for
Advanced Computer Science
NASA Ames Research Center
Moffett Field, CA 94035-1000

Abstract

Compared to the customary column-oriented approaches, block-oriented, distributed-memory sparse Cholesky factorization benefits from an asymptotic reduction in interprocessor communication volume and an asymptotic increase in the amount of concurrency that is exposed in the problem. Unfortunately, block-oriented approaches (specifically, the block fan-out method) have suffered from poor balance of the computational load. As a result, achieved performance can be quite low. This paper investigates the reasons for this load imbalance and proposes simple block mapping heuristics that dramatically improve it. The result is a roughly 20% increase in realized parallel factorization performance, as demonstrated by performance results from an Intel Paragon™ system. We have achieved performance of nearly 3.2 billion floating point operations per second with this technique on a 196-node Paragon system.

1 Introduction

The Cholesky factorization of sparse symmetric positive definite matrices is an extremely important computation, arising in a variety of scientific and engineering applications. Sparse Cholesky factorization is quite time-consuming and is frequently the computational bottleneck in these applications. Consequently, there is significant interest in performing the computation on large parallel machines.

Parallel sparse Cholesky factorization is typically performed using one of two alternative high-level matrix mapping strategies. The first and more traditional approach, a 1-D mapping, distributes rows or columns of the sparse matrix among processors. Unfortunately, this mapping has two major limitations. The first is that it produces interprocessor communication volumes that grow linearly in the number of processors [7]. The communication costs of this approach makes it nonscalable [14], and they also dominate runtimes on all but the smallest parallel machines. The second limitation is caused by the parallel task definition that seems to naturally accompany a 1-D data mapping. A parallel method that breaks the computation into column scaling and column modification tasks suffers from an excessively long critical path [11]. Since the critical path represents a lower bound on parallel runtime, parallel speedups are limited, unnecessarily, by this approach.

The second mapping alternative is a 2-D block mapping, where rectangular blocks of the sparse matrix are mapped to the processors. In such a mapping, we view the machine as a two-dimensional $P_r \times P_c$ processor grid, whose members we denote $P(i,j)$. All blocks in a given block row are mapped to the same row of processors, and all elements of a block column to a single processor column. The advantages of such a mapping are substantial. Communication volumes grow as the square root of the number of processors, versus linearly for the 1-D mapping. The critical path length is also significantly reduced. For a $k \times k$ grid problem, it is $O(k)$ for a 2-D block mapping, versus $O(k^2)$ for a 1-D mapping. These advantages accrue even when the underlying machine has some interconnection network whose topology is not a grid. Several researchers have obtained excellent performance using a block-oriented approach, both on fine-grained, massively-parallel SIMD machines [3] and on coarse-grained, highly-parallel MIMD machines [12].

To define a 2-D block mapping, one must specify the mappings of matrix (block) rows to processor rows and of columns to processor columns. What has been used to date is the 2-D cyclic (also called torus-wrap) mapping: block L_{IJ} resides at processor $P(I \bmod P_r, J \bmod P_c)$. We have observed, however, that a 2-D cyclic mapping (particularly a coarse-grained mapping using large blocks) has a serious limitation: it produces significant load imbalances that severely limit achieved efficiency. On the Intel Paragon system, on which interprocessor communica-

tion bandwidth is quite high, this load imbalance limits its efficiency to a greater degree than communication or want of parallelism.

This paper investigates in some detail the sources of the load imbalance produced by block-oriented methods. After examining the causes of the poor load balance obtained by the cyclic mapping, we propose and investigate a method for improving it. We find that by performing a heuristic remapping of matrix blocks to processors, load imbalance can be mitigated to a point where it is no longer the most serious bottleneck in the computation. Performance results from a block-oriented code run on the Intel Paragon system demonstrate that the block mapping heuristic produces a roughly 20% increase in achieved parallel performance compared with the cyclic mapping. We also find that careful choice of P, the number of processors, can produce substantial performance improvements without remapping; but they are not as large as those produced by the heuristic mapping.

The organization of the paper is as follows. Section 2 provides background on parallel sparse Cholesky, including a discussion of block-oriented factorization and the block fan-out method. Section 3 looks at load balance results from the block fan-out method. Section 4 describes our approach to improving the load balance, and looks at the resulting improvement in load balance and parallel performance. Finally, Section 5 discusses the results.

2 Parallel Sparse Cholesky Factorization

2.1 Computation Structure

The goal of the sparse Cholesky computation is to factor a sparse symmetric positive definite $n \times n$ matrix A into the form $A = LL^T$, where L is lower triangular. We consider only the numeric factorization here; our present codes perform matrix reordering and symbolic factorization sequentially.

In the block factorization approach considered here, matrix blocks are formed by dividing the columns of the $n \times n$ matrix into N contiguous subsets, $N \leq n$. The identical partitioning is performed on the rows. A block L_{IJ} in the sparse matrix is formed from the elements that fall simultaneously in row subset I and column subset J.

The block-oriented sparse factorization is performed with the following pseudo-code:

```
1.  L := A
2.  for K = 1 to N do
3.      L_KK := Factor(L_KK)
4.      for I = K+1 to N with L_IK ≠ 0 do
5.          L_IK := L_IK L_KK^{-1}
6.      for J = K+1 to N with L_JK ≠ 0 do
7.          for I = J to N with L_IK ≠ 0 do
8.              L_IJ := L_IJ - L_IK L_JK^T
```

We will refer to Statement 3, the Cholesky factorization of a diagonal block L_{KK}, as a $BFAC(K,K)$ operation. Similarly, we refer to Statement 5 as a $BDIV(I,K)$ operation, and Statement 8 as a $BMOD(I,J,K)$ operation.

Observe that in a $BMOD()$ operation, the row index of the *destination* block L_{IJ} is the same as the row index of one *source* block, and that the column index is the same as the row index of the other source block. Equivalently, a block L_{IJ} modifies only blocks in block row I or block column I.

2.2 Supernodes

Before discussing parallel sparse factorization, we must first discuss an important concept in sparse factorization, that of a *supernode* [2]. A supernode is a set of adjacent columns in the factor L whose non-zero structure consists of a dense lower-triangular block on the diagonal, and an identical set of non-zeroes for each column below the diagonal block. Supernodes arise in any sparse factor, and they are typically quite large. The regularity in the sparse matrix captured by this supernodal structure can be exploited for a variety of purposes [2, 8, 11, 13]. We exploit it to simplify the internal non-zero structure of blocks of the matrix. Specifically, we create block columns whose member columns belong to the same supernode. As a result, we obtain blocks whose rows are either completely zero or are dense. This regular structure allows the block factorization primitives ($BFAC$, $BDIV$, and $BMOD$) to be implemented efficiently. This regularity can be further increased by performing *supernode amalgamation* [1] on the factor matrix. Amalgamation is a heuristic that merges supernodes with very similar non-zero structures into larger supernodes. We use amalgamation in our experiments.

2.3 Parallel Block-Oriented Sparse Cholesky Factorization

We now discuss parallel block-oriented sparse factorization. As mentioned earlier, the approach we use is the block fan-out method. We give a high-level overview of the method here; detailed descrip-

tions have appeared earlier [12]. Our code is based on the single-program, multiple-data (SPMD), message-passing model of parallel computing. A set of processors execute identical code. All data is private to some processor, and processors communicate via interprocessor messages.

Each block L_{IJ} in the block fan-out method has an owning processor. The owner of L_{IJ} performs all block operations whose destination is L_{IJ} (these include $BFAC(I,J)$ when $I = J$, $BMOD(I,J)$ when $I \neq J$, and all $BMOD(I,J,K)$ operations with L_{IJ} as their destinations). A block L_{IJ} is typically the destination for several block operations.

Interprocessor communication is required whenever a block on one processor modifies a block on another processor. A processor records enough information about each block to allow it to determine when a block has received all block modifications, and to which other processors that block must be sent once the last block modification is performed. A processor also records enough information to allow it to determine which blocks that it owns are modified by a block it receives from another processor. The block fan-out method is entirely "data-driven". A processor acts on received blocks in the order in which they are received from other processors, and it sends blocks it owns to other processors as soon as they are ready to be used as source blocks.

For a sparse matrix, the block fan-out method does not actually perform a 2-D mapping on the entire matrix. Instead, the method splits the matrix into two portions: a *domain* portion and a *root* portion. The domain portion consists of matrix columns corresponding to disjoint subtrees of the elimination tree. (See [10] for a discussion of the elimination tree.) Each subtree is assigned wholly to a single processor so as to evenly distribute the factorization work associated with the domain portion of the matrix among the processors. The domain portion is permuted to the front of the sparse factor matrix, and it is mapped to the processors using a 1-D block-column mapping (although the non-zeroes in the domain portion are still stored as a set of blocks). The root portion of the matrix is mapped to processors using a 2-D mapping. Details on the use of domains are provided elsewhere [12]. The main advantage of using domains is that they significantly reduce interprocessor communication volumes.

2.4 Block Mappings

A crucial issue in any block factorization is the mapping of blocks to processors. We assume for this discussion that the processors P can be arranged as a grid of P_r rows and P_c columns; best performance is obtained when P_r and P_c are both $O(\sqrt{P})$. A *block mapping* is a function:

$$map : \{0..N-1\} \times \{0..N-1\} \rightarrow \{0..P_r-1\} \times \{0..P_c-1\}$$

from blocks L_{IJ} to processors $P(r,c)$ in the processor grid. In its most general form, the mapping is arbitrary: a block can be mapped to any processor in the grid. We now introduce some special classes of block mappings, and show that significant benefits can be obtained from their use.

We define a block mapping to be *Cartesian product* (CP) if

$$map(I,J) = P(mapI(I), mapJ(J)).$$

where

$$mapI : \{0..N-1\} \rightarrow \{0..P_r-1\},$$

$$mapJ : \{0..N-1\} \rightarrow \{0..P_c-1\}$$

are arbitrary. In a CP mapping, a row of blocks is mapped to a row of processors in the processor grid and a column of blocks is mapped to a column of processors. CP mappings are important because they reduce interprocessor communication volumes. As we mentioned earlier, a block L_{IK} can only modify blocks in row I or column I. Any CP mapping guarantees that the modified blocks are only mapped to a single row and a single column of the processor grid. Thus, the maximum number of processors a block must be sent to is $P_r + P_c$, which is $O(\sqrt{P})$.

We say that *map* is *symmetric Cartesian* (SC) if $P_r = P_c$ and $mapI = mapJ$. The most commonly used block mapping is SC, with the cyclic function ($mapI[I] = mapJ(I) = I \bmod P_r$). The resulting mapping is typically referred to as a *2-D cyclic* or *torus-wrap* mapping. While a cyclic mapping is effective at reducing communication (since it is CP), it is unfortunately not very effective at balancing computational load.

3 Block Fan-Out Method Load Balance

We now consider the load balance produced by the cyclic mapping and by SC mappings in general. Experiment and analysis show that the cyclic mapping produces particularly poor load balance; moreover, some serious load balance difficulties must occur for any SC mapping. Improvements obtained by the use of nonsymmetric CP mappings are discussed in the following section.

Table 1: Benchmark matrices.

Name	Equations	NZ in L	Ops to factor (Million)
DENSE1024	1,024	523,776	358.4
DENSE2048	2,048	2,096,128	2,865.4
GRID150	22,500	656,027	56.5
GRID300	90,000	3,266,773	482.0
CUBE30	27,000	6,233,404	3,904.3
CUBE35	42,875	12,093,814	10,114.7
BCSSTK15	3,948	647,274	165.0
BCSSTK29	13,992	1,680.804	393.1
BCSSTK31	35,588	5,272,659	2,551.0
BCSSTK33	8,738	2,538,064	1,203.5

3.1 Benchmark Matrices and Experimental Design

Table 1 lists the sparse matrices we use as benchmark matrices. The list includes two dense matrices (DENSE1024 and DENSE2048), two 2-D grid problems (GRID150 and GRID300), two 3-D grid problems (CUBE30 and CUBE35), and 4 irregular sparse matrices from the Harwell-Boeing sparse matrix test set [4]. The 2-D and 3-D grid matrices are pre-ordered using nested dissection, which gives asymptotically optimal orderings for these problems. The Harwell-Boeing matrices are pre-ordered using multiple minimum degree [9], which is considered the best for most irregular sparse matrices with respect to sequential operation count and fill. Note that the floating-point operation counts are from the best known sequential sparse factorization algorithm. All Mflops measurements presented here are computed by dividing these floating-point operation counts by parallel factorization runtimes.

All of our experiments were performed on an Intel Paragon system at Intel Supercomputer Systems Division in Beaverton, Oregon. All nodes in this system have 32 megabytes of memory, except for two which have 128 megabytes. The machine is running Release 1.2 of Intel's OSF/1 operating system. In this release, message latency is 50 μseconds and message passing bandwidth is at most 75 megabytes per second. For the messages sizes used in our code, the effective bandwidth is roughly 40 megabytes per second.

Our code uses hand-optimized versions of the Level-3 BLAS for almost all arithmetic. The BLAS are used to perform the $BDIV()$ (triangular solution with a matrix of right-hand sides) and $BMOD()$ (matrix multiplication) block operations. Performance for the individual BLAS operations is in the range of 20 — 40 Mflops per processor, depending on the sizes of the arguments.

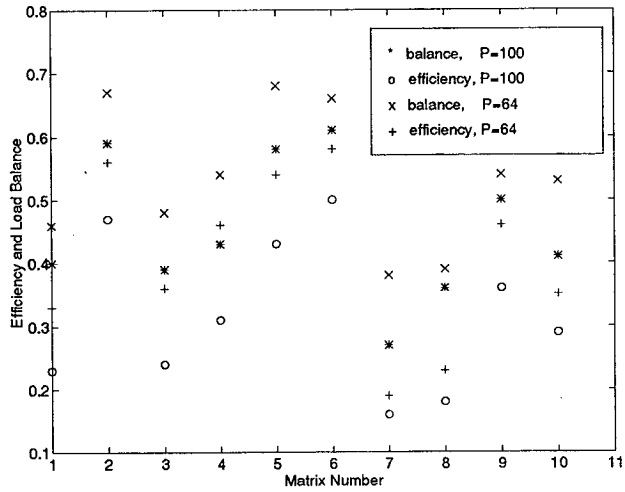

Figure 1: Efficiency and overall balance for the block fan-out method on the Paragon system ($B = 48$).

3.2 Empirical Load Balance Results

We now report on the efficiency and load balance of the block fan-out method. Parallel efficiency is given by

$$\text{efficiency} = t_{seq}/(P \cdot t_{parallel}),$$

where $t_{parallel}$ is the parallel runtime, P is the number of processors, and t_{seq} is the runtime for the same problem on one processor. For the data we report here, we measured t_{seq} by factoring the benchmark matrices using our parallel algorithm on one processor. Although a true sequential algorithm would be slightly faster, we use the parallel algorithm as the baseline because our intent here is to better understand the behavior of this parallel algorithm. Note that we also report absolute parallel performance.

We use the load balance measure

$$\text{overall balance} = work_{total}/(P \cdot work_{max}),$$

where $work_{total}$ is the total amount of work performed in the factorization, P is the number of processors, and $work_{max}$ is the maximum amount of work assigned to any processor (we describe how factorization work is measured later in this section). This ratio gives an upper bound on parallel efficiency:

$$\text{efficiency} \leq \text{overall balance}.$$

Communication time, critical path lengths, and scheduling-induced inefficiencies will generally cause achieved efficiency to be lower than this bound.

Figure 1 shows efficiency and overall balance for our benchmark matrices when factored on 64 and 100 processors. In all our experiments, we choose

$P_r = P_c = \sqrt{P}$. (We do not believe, however, that our proposed heuristic remapping and its benefits are specific to square processor grids.) In all experiments we used a block size of 48. In other words, when we create row and column partitions in order to form blocks, the subset sizes are chosen to be as close to 48 as possible. Recall that column subsets are always subsets of supernodes, so some block columns will have fewer than 48 columns. We choose a block size of 48 because it strikes a reasonable balance between the single-node efficiency of the factorization, which increases with increasing block size, and the amount of concurrency available in the computation, which decreases with increasing block size. Our results would apply for other block size choices as well.

Observe that parallel efficiencies are generally quite low. The best achieved parallel efficiency is 58%, and the worst is 16%. The overall balance bounds are low as well. The best is less than 68% and the worst is 27%. Given the low overall balance bounds, low achieved efficiencies are not surprising. It is clear from the data, however, that the bound is by no means a perfect predictor of achieved efficiencies. Other factors limit performance. Examples include interprocessor communication costs, which we measured at 5% — 20% of total runtime, long critical paths, which can limit the number of block operations that can be performed concurrently, and poor scheduling, which can cause processors to wait for block operations on other processors to complete. Despite these disparities, the data do indicate that load imbalance is an important contributor to reduced parallel efficiency.

To better understand the causes of the load imbalance generated by a cyclic mapping, let us now look at how well the computational load is distributed among groups of processors. Specifically, we consider load imbalance among rows of processors, columns of processors, and diagonals of processors. To explain these measures, we first need to define a few terms. We define $work[I, J]$ to be the amount of work performed on behalf of block L_{IJ} by its owner. This measure would ideally reflect the amount of processor runtime required to perform all operations associated with that block. Runtime is of course difficult to predict without actually performing the corresponding operations. To approximate runtime, we use a work measure that is equal the number of floating point operations performed on behalf of the block plus one-thousand times the number of distinct block operations performed on behalf of the block. The latter term is included to increase the accuracy of the work approximation for matrices that have a large number of small blocks (due to small supernodes). The fixed cost of performing a block operation using small blocks often dominates the overall cost of the operation. The one-thousand op fixed cost was measured from our factorization code.

We define $workI[I]$ to be the aggregate work required by blocks in row I. That is:

$$workI[I] = \sum_{J=0}^{N-1} work[I, J].$$

An analogous definition applies for $workJ$, the aggregate column work.

We define the row balance by

$$\text{row balance} = \frac{work_{total}}{P * work_{rowmax}},$$

where

$$work_{rowmax} = \max_r \frac{\sum_{I:mapI[I]=r} workI[I]}{P_c},$$

This row balance statistic gives the best possible overall balance (and hence parallel efficiency), obtained only if there is no imbalance within a row of processors. The intent is to isolate load balance due to poor distribution of work *across* rows of processors. An analogous expression gives column balance.

We define diagonal balance as:

$$\text{diagonal balance} = \frac{work_{total}}{P * work_{diagmax}},$$

where

$$work_{diagmax} = \max_{0 \le d < P_r} \frac{\sum_{(I,J) \in D_d} work[I, J]}{P_r}$$

and

$$D_d \equiv \{(I, J) : (mapI[I] - mapJ[J]) \bmod P_r = d\}.$$

Note that we use generalized diagonals in this expression. Generalized diagonal d is made up of the set of processors $P(i, j)$ for which $(i - j) \bmod P_r = d$.

Note that the row, column, and diagonal balances are not necessarily independent. In particular, a single highly overloaded processor can affect all three. Note also that due to their coarse nature, these balance measures do not capture all possible causes of load imbalance in the computation. In fact, it is possible for a mapping to have good balances under these measures while still having a poor overall balance. Despite this, the data we present later make it clear that improving these three measures of balance will in general improve the overall load balance.

Table 2 lists the row, column, and diagonal balances with a 2-D cyclic mapping of the benchmark matrices on 64 processors. Recall that a row balance of 0.5 means that if all work assigned to a single row of the processor grid were distributed evenly among the

Table 2: Efficiency bounds for 2-D cyclic mapping due to row, column and diagonal imbalances ($P = 64, B = 48$).

Matrix	Row bal.	Col. bal.	Diag. bal.	Overall bal.
DENSE1024	0.65	0.95	0.69	0.46
DENSE2048	0.80	0.99	0.82	0.67
GRID150	0.78	0.86	0.62	0.48
GRID300	0.85	0.89	0.71	0.54
CUBE30	0.87	0.94	0.77	0.68
CUBE35	0.86	0.94	0.80	0.66
BCSSTK15	0.70	0.69	0.58	0.38
BCSSTK29	0.68	0.75	0.63	0.39
BCSSTK31	0.75	0.95	0.73	0.54
BCSSTK33	0.76	0.89	0.71	0.53

processors within that row, then the overall balance, and hence the efficiency, would be at most 0.5. All three types of imbalance are significant. Diagonal imbalance is the most severe, followed by row imbalance, followed by column imbalance.

We believe these data can be better understood by considering dense matrices as examples (although the following observations apply to a considerable degree to sparse matrices as well). Row imbalance is due mainly to the fact that $workI[I]$, the amount of work associated with a row of blocks, increases with increasing I. More precisely, since $work[I, J]$ increases linearly with J and the number of blocks in a row increases linearly with I, it follows that $workI[I]$ increases quadratically in I. Thus, the processor row that receives the last block row in the matrix receives significantly more work than the processor row immediately following it in the cyclic ordering, resulting in significant row imbalance. Column imbalance is not nearly as severe as row imbalance. The reason, we believe, is that while the work associated with blocks in a column increases linearly with the column number J, the number of blocks in the column *decreases* linearly with J. As a result, $workJ[J]$ is neither strictly increasing nor strictly decreasing.

Note that the reason for the row and column imbalance is not that the 2-D cyclic mapping is an SC mapping; rather, we have significant imbalance because the mapping functions $mapI$ and $mapJ$ are each poorly chosen. These effects would be seen for any rectangular processor grid, that is for any choice of P_r and P_c.

To better understand diagonal imbalance, one should note that blocks on the diagonal of the matrix are mapped exclusively to processors on the main diagonal of the processor grid. Blocks just below the diagonal are mapped exclusively to processors just below the main diagonal of the processor grid. These diagonal and sub-diagonal blocks are among the most work-intensive blocks in the matrix; they receive the most block modifications of any block in the corresponding row. Thus, the diagonal and sub-diagonal processors that receive these blocks receive more work than the other processors. Note that diagonal imbalance is especially acute for the sparse benchmark matrices. Recall that a given block of L may be zero, or may be dense, or (most often) it may have some zero rows and some dense rows. In these problems, the diagonal blocks are the only ones that are guaranteed to be dense. Since we split supernodes into smaller subsets when we form these blocks, the blocks on the first block subdiagonal are often actually subblocks of the diagonal blocks of supernodes, so they too are dense. Moreover, no diagonal block is zero, and relatively few subdiagonal blocks are zero.

The remarks we make about diagonal blocks and diagonal processors apply to *any* SC mapping, and do not depend on the use of a cyclic function $mapI(I) = I \bmod P_r$.

4 Heuristic Remapping

We now describe and evaluate a heuristic to improve load balance. Recall that the primary benefit of using a CP mapping for the block factorization is that it limits communication. Specifically, with a CP mapping a block of the matrix is sent to only one row and one column of processors in the processor grid. The heuristic we use to improve load balance in a block method relies on the simple observation that all SC mappings must suffer from poor diagonal balance, and that among them the 2-D cyclic mappings also suffer from poor row and column balance. We therefore look at nonsymmetric CP mappings, which map rows independently of columns. We choose the row mapping $mapI$ to maximize the row balance, and independently choose $mapJ$ to maximize column balance. Because the symmetry is broken, we do not expect severe diagonal imbalance, since block diagonals are now spread over the whole machine.

Our approach to choosing row and column mappings to improve load balance is to look at the problem as a number partitioning problem[1] [5]. Our goal is to choose a function $mapI$ that minimizes

$$\max_r \sum_{I:mapI[I]=r} workI[I].$$

An analogous goal applies to the choice of $mapJ$.

[1]Number partitioning is a well studied NP-complete problem. The objective is to distribute a set of numbers among a fixed number of bins so that the maximum sum in any bin is minimized.

We shall consider several heuristics for optimizing load balance. All obtain a row mapping as follows:

$mapped[r] := 0,\ \forall r \in \{0..P_r - 1\}$
for each block row I, in some order (see below)
  $minr := r$ for which $mapped[r]$ is minimum
  $mapI[I] := minr$
  $mapped[minr] := mapped[minr] + workI[I]$

This familiar algorithm iterates over all block rows, mapping a block row to the processor row that has received the least work so far ($mapped[r]$ records the amount of work mapped to processor row r). The only difference between the various heuristics we consider is the sequence in which the block rows are mapped. We have experimented with four different sequences, which we now describe.

The **Decreasing Work (DW)** heuristic considers rows in order of decreasing work. This is a standard approach to number partitioning; that small values toward the end of the sequence allow the algorithm to lessen any imbalance caused by large values encountered early in the sequence.

The **Increasing Number (IN)** heuristic considers rows in order of increasing row number. This straightforward approach is included for purposes of comparison; we expect that it will be markedly less effective than the other heuristics.

The **Decreasing Number (DN)** heuristic considers rows in order of decreasing row number. While also straightforward (straightbackward?) we expect it to provide better load balance than the increasing number heuristic. Recall that the amount of work associated with a row generally increases with increasing row number.

The **Increasing Depth (ID)** heuristic considers rows in order of increasing depth in the elimination tree. This refinement of the decreasing number heuristic takes into account the structure of a sparse problem. In a sparse problem, the work associated with a row is more closely related to its depth in the elimination tree than it is to its row number. We therefore guessed that it would be the second best of the four schemes.

We may have intuitive guesses about which of these heuristics will provide the best results. But we do not pretend that this is an exact science. Just as astronomers and their theories are the servants of the sky, we must let the matrices tell us which is best.

4.1 Results

We begin by looking at how the row and column mapping heuristics affect the measures of load balance described above. Table 3 shows the results of applying the heuristics to a single problem, BCSSTK31, on 64 processors. The five rows correspond to the five different mapping heuristics (the row labeled cyclic corresponds to the original cyclic mapping). The numbers in the table give balance results when the corresponding heuristic is applied to both the row and column mapping.

Table 3: Row, column, and diagonal balance for problem BCSSTK31 ($P = 64, B = 48$).

Heuristic	Row bal.	Col. bal.	Diag. bal.	Overall bal.
Cyclic	0.75	0.95	0.73	0.54
Decr. Work	0.99	0.99	0.92	0.76
Inc. Number	0.83	0.96	0.90	0.72
Decr. Number	0.99	0.98	0.93	0.81
Inc. Depth	0.99	0.99	0.96	0.81

As expected, the quality of the row and column mapping depends on the order in which the rows or columns are considered. The decreasing work and increasing depth heuristics produce the best row and column load balances. The increasing row/column number heuristic produces the worst, but is still much better than the symmetric cyclic mapping. Note that all of the heuristics remove the diagonal imbalance. This indicates that independent row and column mappings are extremely effective at removing diagonal imbalance. Note also that the improvements in row, column, and diagonal imbalance together produce dramatic improvements in overall balance.

We now broaden our investigation by including results from all the matrices in the benchmark set, and also by varying the row and column mapping heuristic independently. Table 4 presents this load balance data, giving mean improvement in overall balance over all ten benchmark matrices on 64 and 100 processors. The rows of the table correspond to the five different heuristics for mapping blocks rows to processor rows. The columns of the table correspond to the same five heuristics, applied to the block columns of the matrix. The data reveal that all of our mapping heuristics dramatically improve processor load balance.

Of course, the goal of this heuristic is not to improve load balance, but rather to improve achieved parallel performance. Table 5 presents mean performance improvement numbers from factorizations on 64 and 100 processors of the Intel Paragon system. Comparing the numbers in Table 5 to those in Table 4, one can see that the improvements in performance are not nearly as large as the improvements in load balance. This indicates that after the heuristics are applied, load balance is no longer the most constraining factor limiting performance. The performance improvements are nevertheless quite significant, and they have been obtained at little cost in algorithm complexity.

Table 4: Mean improvement in overall balance ($B = 48$).

Row remapping heuristic	P = 64 Column remapping heuristic					P = 100 Column remapping heuristic				
	CY	DW	IN	DN	ID	CY	DW	IN	DN	ID
Cyclic	0%	18%	17%	21%	17%	0%	19%	23%	22%	21%
Decr. Work	37%	34%	41%	47%	42%	39%	38%	56%	52%	50%
Inc. Number	19%	18%	21%	20%	24%	20%	24%	24%	31%	21%
Decr. Number	39%	37%	43%	43%	47%	41%	36%	50%	50%	49%
Inc. Depth	39%	34%	45%	47%	43%	40%	37%	53%	54%	49%

Table 5: Mean improvement in parallel performance on the Intel Paragon system (B=48).

Row remapping heuristic	P = 64 Column remapping heuristic					P = 100 Column remapping heuristic				
	CY	DW	IN	DN	ID	CY	DW	IN	DN	ID
Cyclic	0%	13%	14%	15%	17%	0%	12%	19%	19%	20%
Decr. Work	21%	14%	18%	21%	19%	20%	16%	21%	19%	20%
Inc. Number	16%	13%	13%	15%	15%	20%	17%	11%	19%	19%
Decr. Number	18%	14%	18%	16%	18%	23%	15%	19%	15%	20%
Inc. Depth	20%	14%	19%	19%	18%	24%	16%	20%	21%	18%

Which heuristic provides the best performance? Several choices provide quite comparable results. In fact, it appears that the most important point is that some remapping for load balance must be done; the particular remapping used is (with a few exceptions) of secondary importance. Indeed, as measured by the row and column load balances, the differences between the various nonsymmetric, heuristically load balanced mappings is much smaller than the difference between any of them and the symmetric cyclic mapping. Recall that the most significant contributor to load imbalance is the diagonal imbalance, and all the nonsymmetric mapping strategies remove this imbalance.

4.2 Alternative Heuristics

In addition to the heuristics described so far, we also experimented with two other approaches to improving factorization load balance. The first is a subtle modification of the original heuristic. It begins by choosing some column mapping (we use a cyclic mapping). This approach then iterates over rows of blocks, mapping a row of blocks to a row of processors so as to minimize the amount of work assigned to any one *processor*. Recall that the earlier heuristic attempted to minimize the aggregate work assigned to an entire row of processors. We found that this alternative heuristic produced further improvements in load balance (typically 10-15% better than that of our original heuristic). Unfortunately, realized performance did not improve. This result partially confirms our earlier claim that load balance is not the most important performance bottleneck once our original heuristic is applied.

The other approach we considered is significantly simpler than any of the approaches described so far. This approach reduces imbalance by performing cyclic row and column mappings on a processor grid whose dimensions $P_c \times P_r$ are relatively prime. The relatively prime dimensions eliminate diagonal imbalance (the most significant form of imbalance), since cyclic row and column mappings now scatter diagonal matrix blocks among all processors in the grid. We found that we could obtain significantly higher performance than that reported earlier for symmetric cyclic mappings if we used one *fewer* processor (e.g., 63 processors instead of 64, and 99 processors instead of 100). In both cases, one fewer processor produces relatively prime grid dimensions. The improvement in performance was somewhat lower than that achieved with our earlier remapping heuristic (17% and 18% mean improvement on 63 and 99 processors versus 20% and 24% on 64 and 100 processors).

4.3 Results for Larger Machines

We now present results from larger sparse problems factored on a larger machine. Table 6 gives information about the larger sparse problems. Problem DENSE4096 is a dense matrix. Problem CUBE40 is a regular 3-d cube. It is ordered using nested-dissection. Problem COPTER2 is a model of a helicopter rotor blade, given to us by Horst Simon of NASA Ames. It is ordered using multiple minimum degree. Problem 10FLEET comes from the linear programming formulation of an airline fleet assignment problem. It was

Table 6: Large benchmark matrices.

Name	Equations	NZ in L	Ops to factor (Million)
DENSE4096	4,096	8,386,560	22,915
CUBE40	64,000	21,408,189	23,084
COPTER2	55,476	13,501,253	11,377
10FLEET	11,222	4,782,460	7,450

given to us by Roy Marsten of Delta Airlines. It is also ordered using multiple minimum degree. We also include performance numbers for two matrices from our previous set, CUBE35 and BCSSTK31.

Table 7 shows our results for these problems on 144 and 196 processors. The table shows performance using a cyclic mapping and using our mapping heuristic (increasing depth on rows and cyclic on columns). Note that our heuristic again produces a roughly 20% performance improvement over a cyclic mapping.

5 Discussion

We have demonstrated that the performance of parallel block-oriented sparse factorization can be affected significantly by the mapping of blocks to processors. With the traditional 2-D cyclic mapping, load imbalance limits performance. Our mapping heuristics improve load balance to the point where it appears to no longer be the most constraining performance bottleneck. Two questions that arise are: (i) what is the most constraining bottleneck after our heuristic is applied, and (ii) can this bottleneck be addressed to further improve performance?

One potential remaining bottleneck is communication. Instrumentation of our block factorization code reveals that on the Paragon system, communication costs account for less than 20% of total runtime for all problems, even on 196 processors. The same instrumentation reveals that most of the processor time not spent performing useful factorization work is spent idle, waiting for the arrival of data.

One possible explanation for this idle time is that there simply is not enough work to be performed in parallel. This does not appear to be the case: an analysis of the critical path of the factorization computation [11], which gives a coarse bound on the amount of parallelism in the problem, reveals that while these problems certainly do not admit a large surplus of concurrency, there should be enough to keep the processors occupied. Critical path analysis for problem BCSSTK15 on 100 processors, for example, indicates that it should be possible to obtain nearly 50% higher performance than we are currently obtaining. The same analysis for problem BCSSTK31 on 100 processors indicates that it should be possible to obtain roughly 30% higher performance.

Another possibility is that poor scheduling of the block operations limits performance. It is possible that low-priority block operations delay higher priority block operations, causing processors that are waiting for the results of the high-priority operations to sit idle. We hope to investigate the use of dynamic scheduling techniques that are more sensitive to some measures of priority of tasks than is the purely "data-driven" approach used in the block fan-out method.

We should mention that as part of this work we also explored mappings whose goal was to reduce interprocessor communication volumes. Our approach was loosely based on the subtree-to-subcube mapping scheme originally used for column-oriented sparse factorization [6]. Here, the processor set is divided recursively among the subtrees in the elimination tree. Matrix columns belonging to these subtrees are mapped exclusively to the appropriate submachines. In our block-oriented code, rather than dividing *processors* among subtrees, we instead divided *processor-columns* of the processor grid among subtrees. We found that communication volumes were reduced by as much as 30%. Unfortunately, load balancing became more difficult. The best load balance we were able to obtain using a rough subtree-to-subcube mapping on columns and a heuristic mapping on rows was roughly the same as the load balance with a pure cyclic mapping. Since communication costs were not a significant performance bottleneck on the Paragon system, the resulting factorization performance was lower than that produced by our heuristic mappings.

One other approach we considered for improving load balance was to vary the block size. Intuitively, it would appear that the factorization computation can tolerate large blocks towards the beginning of the factorization, when there is a significant amount of concurrent work available to hide load imbalances caused by the large blocks. Small blocks would be more appropriate towards the end of the computation. We discovered that this intuition is actually incorrect. Varying the block size between the early stages of the computation and the later ones has no effect on load imbalance; and it reduces the amount of parallelism available in the problem. It is possible, however, to improve the load balance by choosing the block size based on the processor row/column to which the block is mapped. We succeeded in improving performance using this approach, but the improvement was not nearly as large as it was using the block remapping strategies we have described.

One final issue that merits further investigation is whether the sparse factorization approach proposed here may actually provide higher performance for

Table 7: Performance (Mflops) for larger benchmark problems on 144 and 196 nodes, using a cyclic mapping, and using an increasing depth row remapping and cyclic column mapping.

	P = 144			P = 196		
Matrix	Performance: cyclic	Performance: heuristic	Performance improvement	Performance: cyclic	Performance: heuristic	Performance improvement
CUBE35	1788	2207	23%	2019	2456	22%
CUBE40	2093	2384	14%	2515	3187	27%
DENSE4096	3587	4156	16%	4489	5237	17%
BCSSTK31	1161	1322	14%	1361	1709	26%
COPTER2	1693	1779	5%	1959	2312	18%
10FLEET	2027	2246	11%	2488	2722	9%

dense problems than is currently obtained by specialized dense factorization methods [15] that use cyclic mappings.

References

[1] Ashcraft, C.C., and Grimes, R.G., "The influence of relaxed supernode partitions on the multifrontal method", *ACM Transactions on Mathematical Software*, 15(4): 291-309, 1989.

[2] Ashcraft, C.C., Grimes, R.G., Lewis, J.G., Peyton, B.W., and Simon, H.D., "Recent progress in sparse matrix methods for large linear systems", *International Journal of Supercomputer Applications*, 1(4): 10-30, 1987.

[3] Conroy, J., Kratzer, S., and Lucas, R. "Data parallel sparse LU factorization", Proceedings, *Seventh SIAM Conference on Parallel Processing for Scientific Computing*, to appear.

[4] Duff, I.S., Grimes, R.G., and Lewis, J.G., "Sparse Matrix Test Problems", *ACM Transactions on Mathematical Software*, 15(1): 1-14, 1989.

[5] Garey, M., and Johnson, M., *Computers and Intractability: A Guide to the Theory of NP-Completeness*, 1982.

[6] George, A., Heath, M., Liu, J., and Ng, E., "Solution of sparse positive definite systems on a hypercube", *Journal of Computational and Applied Mathematics*, 27(1): 129-156, 1989.

[7] George, A., Liu, J. and Ng, E., "Communication results for parallel sparse Cholesky factorization on a hypercube", *Parallel Computing*, 10: 287-298, 1989.

[8] Gilbert, J., and Schreiber, R., "Highly parallel sparse Cholesky factorization", *SIAM Journal on Scientific and Statistical Computing*, 13: 1151-1172, 1992.

[9] Liu, J., "Modification of the minimum degree algorithm by multiple elimination", *ACM Transactions on Mathematical Software*, 11: 141-153, 1985.

[10] Liu, J., "The role of elimination trees in sparse factorization", *SIAM Journal on Matrix Analysis and Applications*, 11:134-172, 1990.

[11] Rothberg, E., *Exploiting the memory hierarchy in sequential and parallel sparse Cholesky factorization*, Ph.D. thesis, Stanford University, January, 1993.

[12] Rothberg, E., and Gupta, A., "An efficient block-oriented approach to parallel sparse Cholesky factorization", *Supercomputing '93*, p. 503-512, November, 1993.

[13] Rothberg, E., and Gupta, A., "An evaluation of left-looking, right-looking, and multifrontal approaches to sparse Cholesky factorization on hierarchical-memory machines", Technical Report STAN-CS-91-1377, Stanford University, 1991.

[14] Schreiber, R., "Scalability of sparse direct solvers". In Alan George, John R. Gilbert, and Joseph W.H. Liu, editors, *Graph Theory and Sparse Matrix Computation*, pp. 191–209. The IMA Volumes in Mathematics and Its Applications, Volume 56, Springer-Verlag, New York, 1993.

[15] Van De Geijn, R., *Massively parallel LINPACK benchmark on the Intel Touchstone Delta and iPSC/860 systems*, Technical Report CS-91-28, University of Texas at Austin, August, 1991.